

510.84

I l6r

no.1042-

1052

1980

Incompl.

c.3

The person charging this material is responsible for its return to the library from which it was withdrawn on or before the **Latest Date** stamped below.

Theft, mutilation, and underlining of books are reasons for disciplinary action and may result in dismissal from the University.

To renew call Telephone Center, 333-8400

UNIVERSITY OF ILLINOIS LIBRARY AT URBANA-CHAMPAIGN

MAY 04 1993

L161—O.1096

THE EXECUTE STATEMENT:
DESIGN, EXAMPLES, AND IMPLEMENTATION ALGORITHMS

by

Martin S. McKendry
Roy H. Campbell

December 1980



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

UIUCDCS-R-80-1044

THE EXECUTE STATEMENT:
DESIGN, EXAMPLES, AND IMPLEMENTATION ALGORITHMS

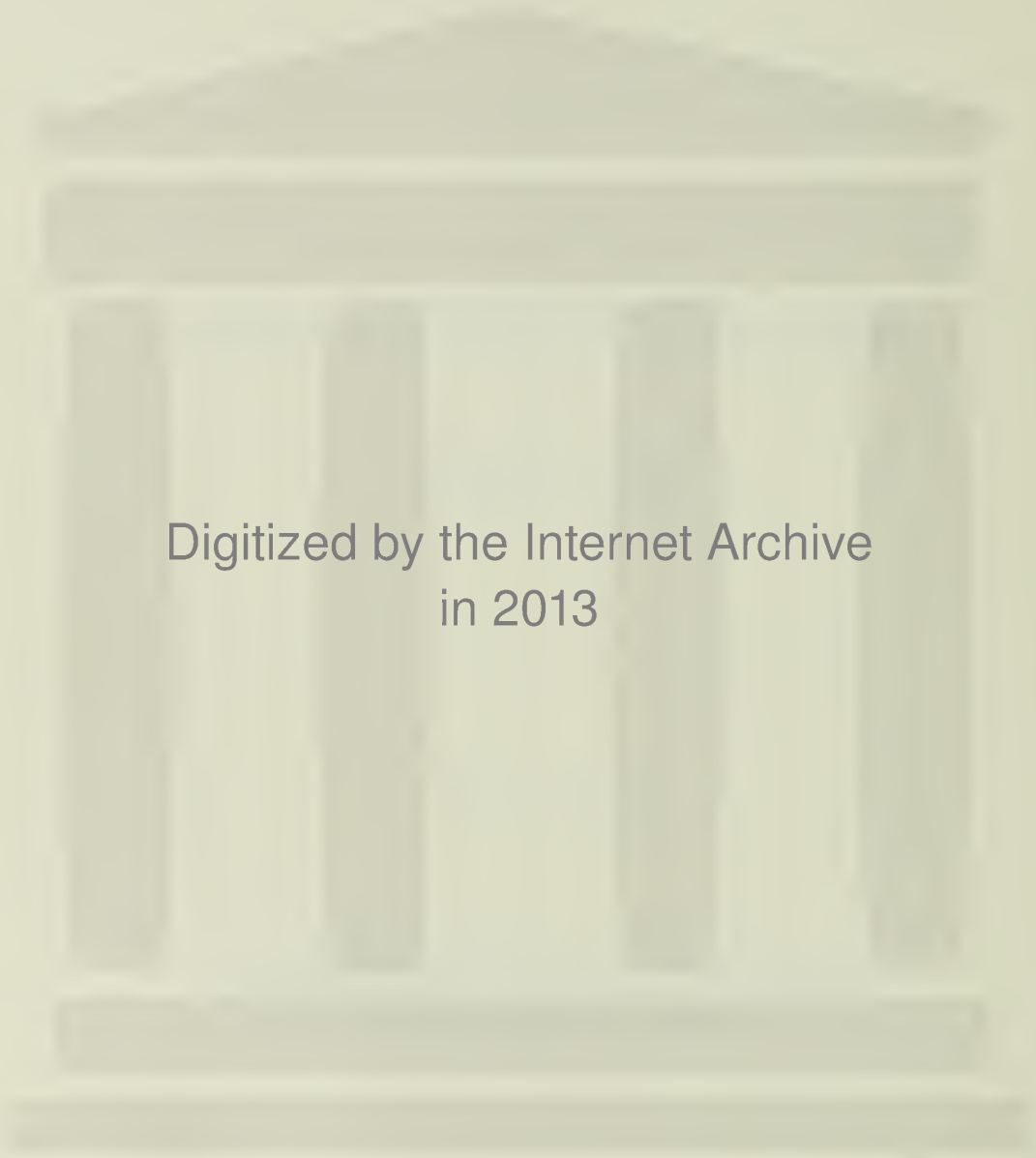
by

Martin S. McKendry
Roy H. Campbell

December 1980

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
URBANA, ILLINOIS 61801

Research supported in part by NASA Project NSG 1471.



Digitized by the Internet Archive
in 2013

<http://archive.org/details/executestatement1044mcke>

ABSTRACT

The Execute statement is a language mechanism that manages the interface between levels of processes in level structured operating systems. It is a high level means of controlling instantiation and management of processes whose number and requirements are unknown when the operating system is compiled. The statement facilitates implementation of services such as memory management, interrupt management, and synchronization. Together with language capabilities, the Execute statement extends the benefits of high level languages to entire operating systems. This is done efficiently, and without additional hardware. The Execute statement makes explicit the interfaces between conceptual levels of abstraction, with inter-level protection enforced by static language constraints.

1 Introduction.

Current high-level operating system languages only permit construction of systems in which all program code is compiled prior to loading of the operating system [Brinch-Hansen, 75] [Wirth, 77] [Campbell & Kolstad, 80]. Operating systems written in these languages use the 'kernel' approach -- they consist of a kernel, coded in assembly language, that manages low-level functions such as processor scheduling and synchronization, and a set of higher level processes, compiled as a single unit in the operating system language, that handle functions such as device management and disk file organization. As a result, the languages are used mainly for dedicated applications, where a complete specification of all the programs that will ever run in the system is available when the system is built.

In many systems, however, complete specifications are not known when the system is built. The most common example is an interactive or batch system, in which users prepare programs then execute them. The programs are prepared after the system is commissioned -- consequently, they cannot be run by an operating system coded in a current high level operating system language unless the entire operating system is recompiled.

User programs represent an extra 'level' of software above the basic operating system. This concept can be generalized [Dijkstra, 68]: many primitive functions, such as synchronization, memory management, or logical I/O, are at the system's lower levels, providing services to its higher levels. This paper presents a high level language construct, the Execute statement, that manages the interface between the levels of level structured systems. The Execute statement allows dynamic addition and deletion of processes, recognition of service requests between operating system levels, explicit

assignment of a processor to a process, and specification of information passing, sharing, and protection between levels. Thus, arbitrary programs can be loaded and executed, regardless of their time of compilation. The Execute statement retains static protection that arises from strong typing in high level languages such as Pascal [Jensen & Wirth, 74]. Dynamic protection is based on language capabilities [McKendry & Campbell, 80] which are used to control access between levels.

The next chapter of this paper sets terminology for discussion of level structured systems. Chapter 3 characterizes the interface between system levels, then Chapter 4 presents the Execute statement with examples of its use. Chapter 5 discusses implementation algorithms.

2 Operating System Structures.

Results of recent work in operating systems indicate that level structures aid problem decomposition, verification, proving, and error containment [Dijkstra, 68] [Liskov, 72]. A level structured system is viewed as a series of abstract machines or levels of abstraction [Dijkstra, 68] [Horning & Randell, 73]. At the most primitive level (level 0), the abstract machine corresponds to actual hardware. Level 1 implements services (such as scheduling and concurrency) not available in hardware. In general, level i (called a server) supports level $i+1$ and implements services for use by higher levels k : ($k > i$) (called users). Level i can request services implemented at lower levels j : ($j < i$), unless those services have been masked at intervening levels [Neumann et. al., 77].

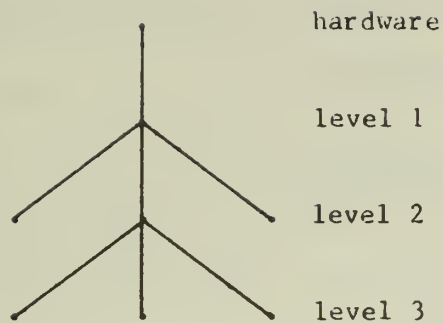


Figure 1: Process Structuring

In this model, each level of abstraction is represented by a disjoint set of processes, said to implement the level (Figure 1). Each node in Figure 1 represents a process that has its own state vector and address space. Level i creates level $i+1$ by instantiating a set of processes to implement level $i+1$. A process that creates other processes is called the parent of the created processes. The created processes are called descendants or sub-processes of the parent. A parent is unable to distinguish between an immediate child and more distant descendants -- transparency is such that they all appear to be a single process. By lifting the constraint that only one process at each level may have children, possible process structures extend to general trees. These are called execution trees.

The Execute statement allows a parent to assign its processor to a child. Since the processor leaves the parent process, only one process at a time is running. Execution is said to be at this process. The path between it and the root of the tree is called the active branch. While executing, a process can request any service offered on the active branch. The service will be provided by the nearest ancestor offering it.

The Execute statement enables dynamic addition and deletion of processes from an execution tree, allocation of resources to those processes,

assignment of the processor to them, and recognition of requests for services implemented by ancestors.

3 Problem Specification.

3.1 Requirements.

There are three requirements for managing the interface between levels: adding processes, assigning them a processor, and providing them with services. Provision of services further subdivides into two parts: recognizing requests for service and, to satisfy those requests, enabling access between process address spaces.

Consider the specific interface shown in Figure 2. In this example, a process at level n needs to load another process at level $n+1$, then run that process and provide the logical I/O service to it.

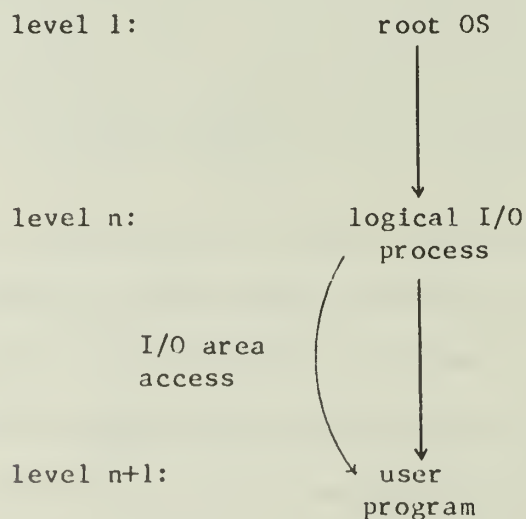


Figure 2: A Process Interface

To fulfil its function, the server process at level n must be able to

share the processor between user processes at level $n+1$. In turn, user processes must be able to relinquish the processor to request services. Provision of services requires that a server process be able to access areas of a user's address space. Thus, at least while a service is being performed, some area must be common to both a server and a user. The essence of the interface between operating system levels is this common area and the processor sharing between server and user processes.

3.2 Protection.

One of the features of high level languages is the protection that they provide [Morris, 73]. In managing the interface between levels of an operating system, it is desirable that this protection be retained. It would be violated if, for example, a process could access the entire address spaces of its descendants -- because then it could alter code areas, raising the possibility of access to illegal services, critical operating system code, and secure data areas.

To protect against this, the Execute statement constrains access to be in accordance with data typing and language capabilities. Capabilities are used for access to the common areas shared between levels -- access must conform to the operations allowed on the item types of the capabilities. A secure loader manages these capabilities, controlling access by checking their types, establishing the safety of access, granting access when a service is requested, and revoking it again once a service has completed.

4 The Execute Statement.

An example Execute statement is:

```
EXECUTE user UNTIL
    service_name_1: service_procedure_1 (parameter_list_1);
    service_name_2: service_procedure_2 (parameter_list_2);
END (* execute *)
```

The variable 'user' is a process state vector (psv) capability. Psv's contain the information needed to represent a process -- here, the user is the child of the process containing the Execute statement (the server). Psv capabilities are managed [McKendry & Campbell, 80] in the root operating system (in terms of Figure 2), and so cannot be copied by general processes. Nor can they be used to alter the contents of process state vectors.

```
TYPE psv = RECORD      (* process status vector *)
    execute_link      : ^psv;
    stack_pointer     : word;
    machine_regs      : register_record;
    page_map          : page_map_record;
    code_file         : code_file_record;
END;                  (* psv *)

psv_cap = CAPABILITY psv; END;    (* no read, no write *)
```

Service names specify the services offered by a process. Together, these services are called a service list. Processes are constrained to at most one Execute statement, so the loader can bind process to services at load-time. Service procedures are invoked when a service is requested.

When the Execute statement is encountered, the processor encountering it is immediately assigned to the user process. Execution leaves the server, staying with the user (at least, from the server's point of view) until the user (or one of its children) requests a service specified in the service list. Transparent to the server are both the specific process that makes the

request and requests to the server's ancestors.

In a server, formal parameters to a service procedure are capabilities. Actual parameters are capability variables that are declared by the server program. The Execute mechanism sets these pointing into the user's address space when a user requests a service. They are revoked when the user is re-Executed, avoiding possible 'dangling pointer' problems -- pointers to non-existent or illegal data areas. Thus, the server has access to the memory space of the user for only as long as is needed to provide the service.

To a user, service requests may appear to be hardware functions (e.g., page faults), language properties (e.g., logical I/O), or the result of explicit conventions introduced by a server (e.g., a message system). Services that result from explicit conventions are called utility services. A utility service is used in the same way as a procedure call and requires declaration by a user (like a 'forward' procedure declaration). Services that are language properties are called language services, and services that are hardware or operating system functions are called system services.

For all services, the types of service procedure parameters are checked when a server is loaded and, in the case of utility services, when a user is loaded. Service parameters in a user must be declared as VAR parameters (call by reference), because they are pointers into the user's memory space. In a server, service parameters are declared as capabilities for the parameters declared by users. These capabilities are managed by the loader, which uses explicit type matching at load time to form a 'symbolic' capability manager. The loader uses information in the Execute statement itself and in the declaration by users of utility services to check that service parameters are properly declared.

Process loading is implemented as a utility service. Because the loader contains a manager for `psv_caps`, it is able to create and initialize process state vectors. To load a user process, a server declares a `psv_cap` and requests the load service, specifying the secondary storage location where the binary code file for the user process is to be found:

```
VAR user: psv_cap;
  :
  :
  load (user, <code file specification>);
```

4.1 Example 1: Logical I/O.

The first example of the Execute statement shows a server process for the language service depicted in Figure 2: logical I/O. In this example, service parameter types -- '`io_area`' and '`io_cap`' -- are declared in the logical I/O server and checked by the loader. A file object, similar to the port object discussed in [McKendry & Campbell, 80], controls buffers and access to physical files. It is accessed via a file capability.

The I/O process accesses files by exchanging buffers with physical I/O processes -- it copies the contents of buffers to and from the I/O areas of descendants. Structurally, it is an infinite loop, repeatedly checking to see whether any input has completed, then selecting a child to Execute. Children are selected on a round-robin basis. The example does not show their loading.

```
(* global declarations *)
```

```
TYPE buffer = RECORD
    size: INTEGER;
    data: ARRAY [1..79] OF CHAR;
END; (* buffer *)
```

```
    buff_cap = CAPABILITY buffer; read, write END; (* managed 'symbolically'
                                                    by the loader *)
```

```
PROCESS logical_io;      (* the logical I/O server process *)
```

```
TYPE io_area = RECORD
    size: integer;
    data: ARRAY [1..79] OF CHAR;
END;
```

```
    io_cap = CAPABILITY io_area; read, write; imported END;
```

```
child_rec = RECORD
    status    : (ready, waiting);
    ps_vector: psv_cap;
    io_vector: io_cap;
    io_buffer: buffer_cap;
END: (* child_rec *)
```

```
VAR child: ARRAY [1..n] OF child_rec;
    file : file_cap;
```

```
PROCEDURE initiate_input (VAR where: io_cap);
    child [i].status := waiting;
    file^.receive (child [i].io_buffer);
    (* This is a call on an object. The object gets the buffer
    * (from a physical I/O process) and sets 'io_buffer' pointing
    * to the buffer. The logical i/o process does not wait for completion. *)
```

```
PROCEDURE initiate_output (VAR where: io_cap);
    file^.get_buffer (child [i].io_buffer);
    WITH child [i] do
        FOR j := 1 to io_vector^.size do
            buffer_cap^.data[j] := io_vector^.data [j];    (* copy data *)
        buffer_cap^.size := io_vector^.size;
        file^.send (buffer_cap);    (* transmit buffer to file object *)
```

```

PROCEDURE check_completions;
    (* check for input completions, and make data transfer *)
    FOR k := 1 to n do
        WITH child [k] DO
            IF status = waiting AND buffer_cap <> NIL THEN
                FOR j := 1 TO buffer_cap^.size DO
                    io_vector^.data [j] := buffer_cap^.data [j];    (* copy *)
                    io_vector^.size := buffer_cap^.size;
                    file^.release (buffer_cap);
                    status := ready;
                END FOR;
            END IF;
        END WITH;
    END FOR;

    (* main line of logical I/O server process *)

DO FOREVER
    FOR i := 1 TO n DO
        IF child [i].status = ready THEN
            EXECUTE child [i].ps_vector UNTIL
                io_in : initiate_input (child [i].io_vector);
                io_out: initiate_output (child [i].io_vector);
                END; (* execute *)
            check_completions;
        END IF;
    END FOR;
END. (* logical I/O process *)

```

When the I/O server Executes child [i], execution passes to that child, staying there until the child (or one of its descendants) requests one of the services 'io_in' or 'io_out'. At that time, the capability child [i].io_vector is set to point to the io_area of the requesting process. The I/O server uses this capability to provide the service, then, when child [i] is again Executed, child [i].io_vector is revoked and execution returns to the requesting process.

4.2 Example 2: Buffer Manager.

The buffer manager is a capability manager [McKendry & Campbell, 80] that maintains a pool of buffers for use by its descendents. It illustrates utility services.

The buffer manager allocates buffers to users when service requests are received. Since it issues user processes with buffer capabilities, the

buffer manager must access user space with a capability for a capability ('b_cap_cap'). The example assumes the same buffer declaration as is used in Example 1:

```

PROCESS buffer_manager;      (* buffer request and release server *)

TYPE b_cap      = capability buffer; read, write END;
                                (* managed by this process *)
    b_cap_cap = capability b_cap; read, write; imported END;
                                (* managed by the 'system' *)

VAR request_cap: b_cap_cap;
    child:      psv_cap;

PROCEDURE buff_request (VAR buff: b_cap_cap);
    buff^ := pointer to a buffer from the pool;

PROCEDURE buff_release (VAR buff: b_cap_cap);
    release buff^ to the pool;
    buff^ := NIL;                (* revoke access *)

EXECUTE child UNTIL
    b_req: buff_request (request_cap);
    b_rel: buff_release (request_cap);
END; (* execute *)

END. (* buffer manager *)

```

A descendant of the buffer manager declares a buffer capability, then declares the services and uses them:

```

VAR b_var: b_cap;                (* needed to use services *)

PROCEDURE buff_request (VAR b: b_cap); SERVICE [b_req]; (* declare services *)
PROCEDURE buff_release (VAR b: b_cap); SERVICE [b_rel];

buff_request (b_var);                (* use services *)
buff_release (b_var);

```

5 Implementation Algorithms.

To implement the Execute statement, cooperation between the loader and the compiler is necessary. The loader, providing symbolic capability management, checks that parameters for service procedures are capabilities for the variables passed by users. The compiler assumes that this checking is correct, and that the loader declares psv's and psv capabilities properly. Since the type-checking algorithms are similar to those used by compilers, they are not discussed further. This chapter discusses algorithms for the Execute statement -- particularly language and utility services -- in some detail.

The Execute statement has two main purposes: First, it controls process context switching -- as the result of an explicit request when a server Executes a user, and as the result of an implicit request when a user requests a service. Second, it performs the 'bookkeeping' operations needed to set a server's service capabilities pointing to the data areas in the user's space when a service is requested and revoke those capabilities when the service is complete.

The explanation in this chapter assumes the reader to be familiar with Pascal -- in particular, call-by-value, call-by-reference (by VAR in Pascal), and their underlying implementation representations. When a request is made, pointers must be coerced to capabilities. This is assumed to be automatic. Variables that were declared as capabilities are still called capabilities at run-time, although they may be implemented as pointers.

5.1 Data Structures.

The execution configuration depicted in Figure 3 (the solid lines are where a parent has actually Executed a child) is shown as a 'snapshot' of data structures in Figure 4. Here, the 'current process' pointer indicates that execution is at process C (stacks grow downward).

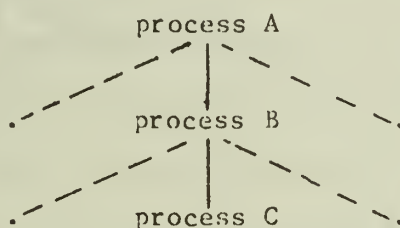


Figure 3: Execution Configuration

The execute stack records the current state of 'Execution'. It contains a pointer to each process on the active branch. If a process is not on this branch, either it has requested a service, one of its descendants has requested a service of one of its ancestors, or it has never been Executed. For a process not on the active branch, the 'execute link' of its psv record shows whether it has Executed a child, indicating the process to which execution will actually return when this process is re-Executed.

5.2 Service Request Implementation.

Whereas a service request can be coded in the source language like a procedure call, its actual machine code representation is as a pair: (level difference, service number). The level difference is the number of levels between the user and server, and the service number is a 'case index' into the service list in the server's Execute statement. The loader sets these values at load time -- it retains a map of all the services offered through the

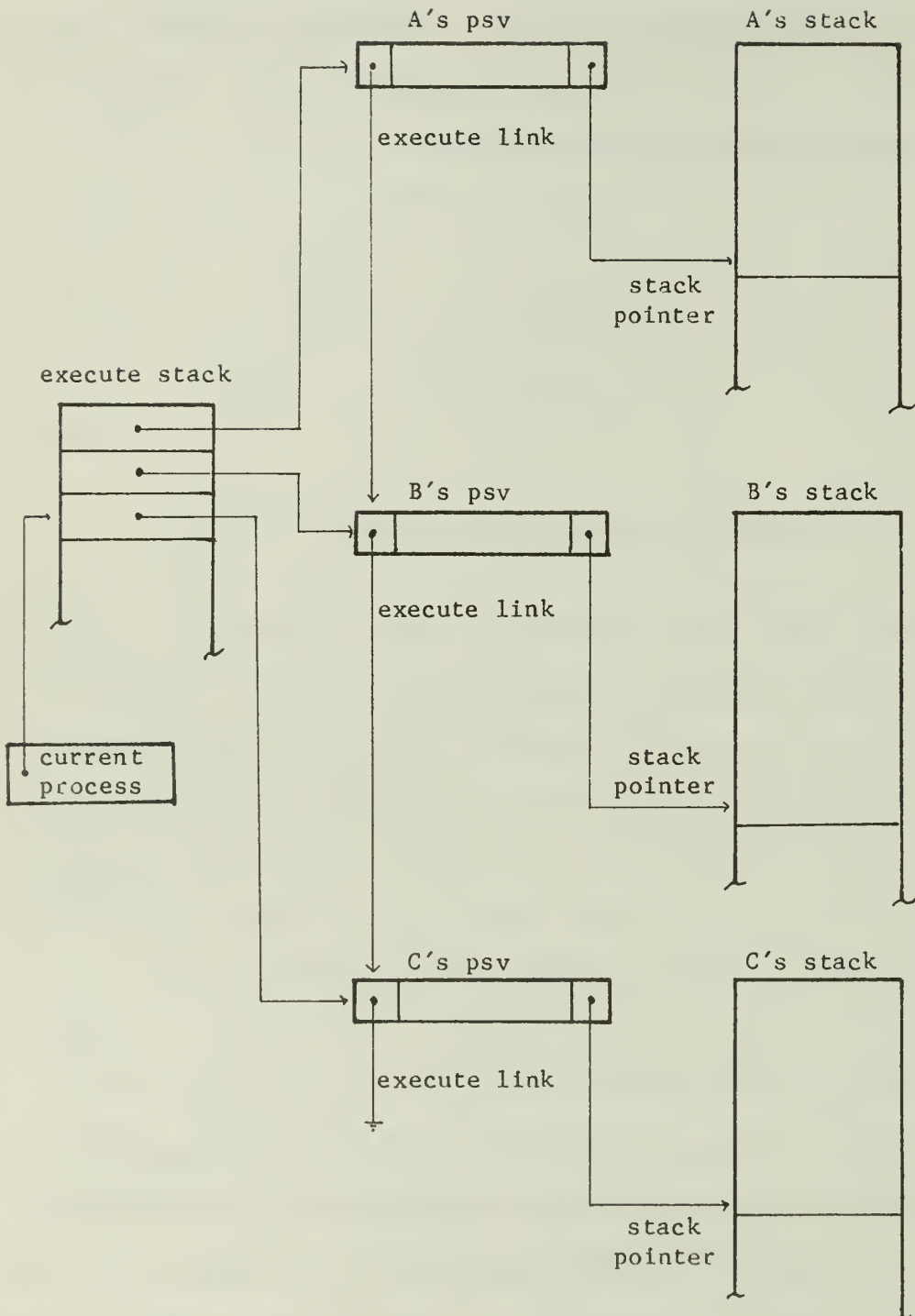


Figure 4: Basic Execute Data Structures

execution tree.

When a service request is encountered, several changes occur in the snapshot of Figure 4. Initially, machine code for a service call is the same as for a procedure call -- procedure parameters are built on the user's stack. The procedure call is not made, however. Instead, the service number is pushed onto the stack (shown in Figure 5), and the processor switches contexts from the user to the server process. To make the context switch, the execute stack is popped according to the level difference, the processor transfers to the process now at the top of the execute stack (the server), and the server's execute link is set to NIL. Next, the service number, still on the user's stack, is popped and used to index the service list in the server's Execute statement. Code is then executed to build the actual parameters for the service procedure.

Data structures at this point are shown in Figure 6. Here, the user's stack contains pointers to the service procedure's actual parameters -- because they were declared as VARs. The server's stack contains pointers to the actual parameters in the server's space -- because, according to the Execute statement they are capabilities, and according to the use of capabilities as parameters [McKendry & Campbell, 80], they too were declared as VARs.

Before the server procedure is called, two bookkeeping operations are needed. First, the parameter pointers on the user's stack are popped and copied into the server's actual parameters (establishing the link marked 4 in Figure 7). Then, the pointers on the server's stack are pushed onto the user's stack (establishing the link marked 3) -- these pointers on the user's stack are visible to neither the user nor the server. They are needed later to automatically revoke the server's capability parameters.

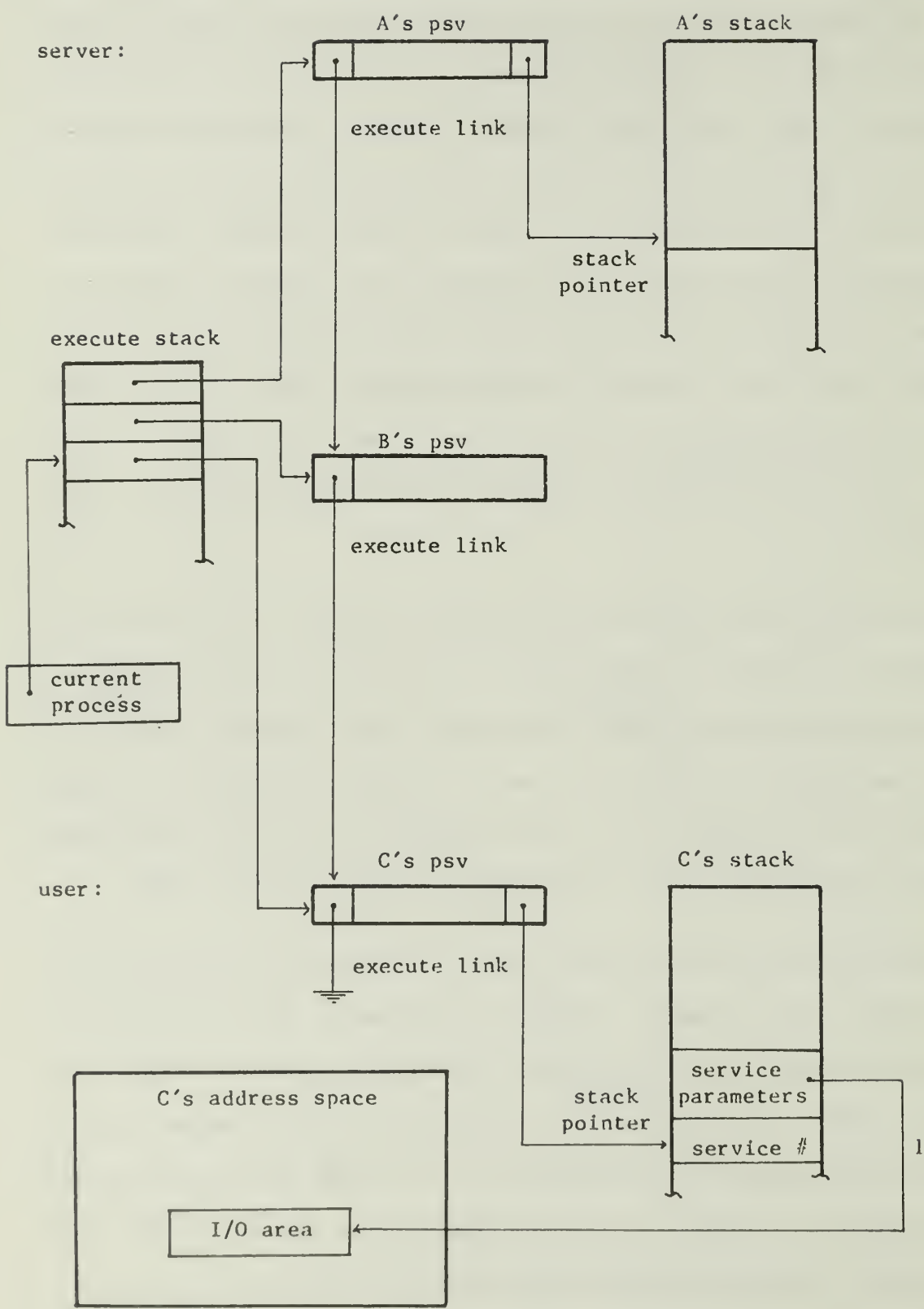


Figure 5: User's Parameters

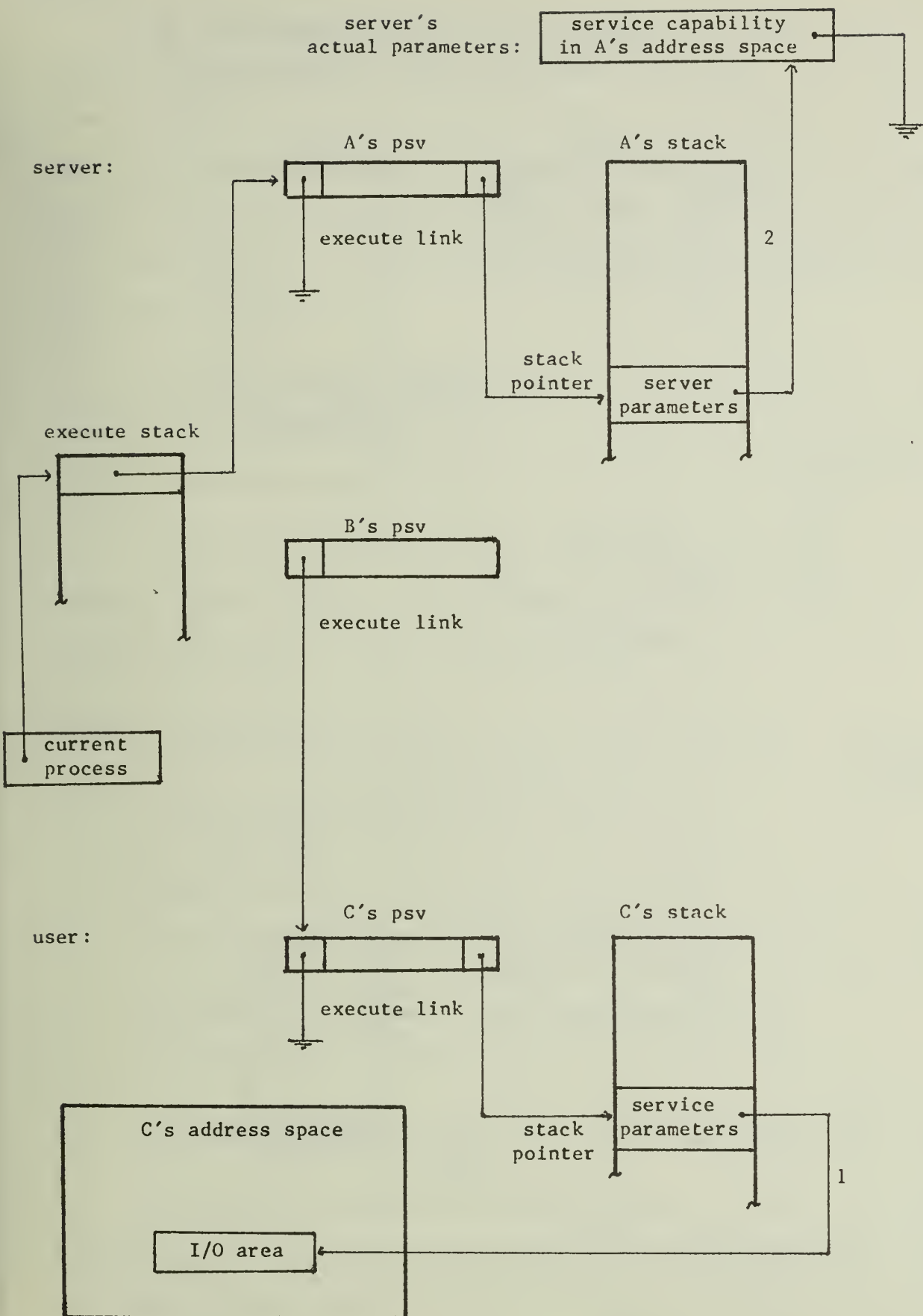


Figure 6: User's and Server's Parameters

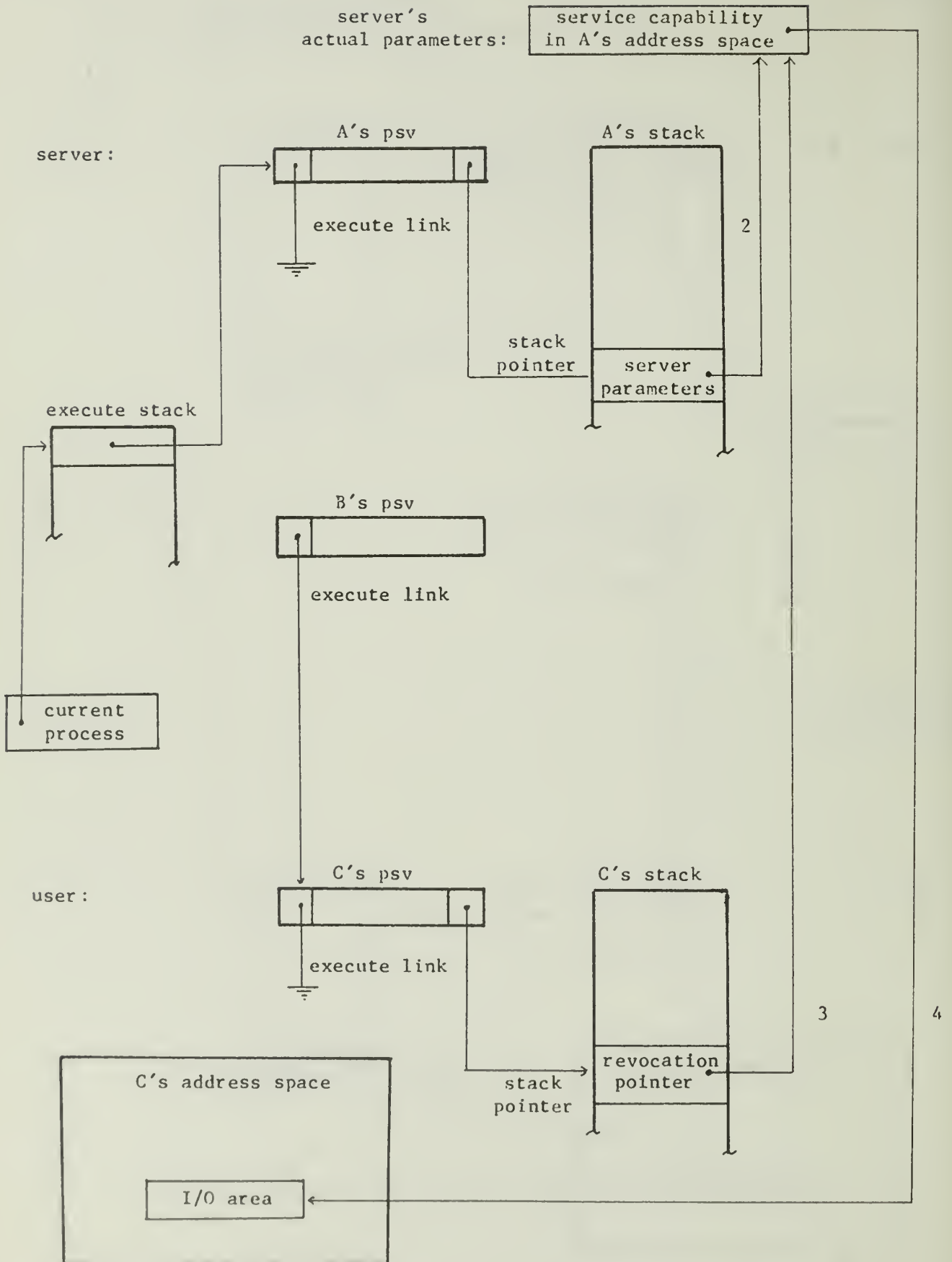


Figure 7: Service Capability and Revocation Pointer

With the situation now as shown in Figure 7, the request sequence is complete and the service procedure is about to be invoked. The execute link from the immediate parent (B) to the user (C) is still in place; it is used when the server re-Executes the user. At that time, execution returns to C, since it was C that requested the service. This supports the transparency described earlier -- a server cannot distinguish the particular descendant that makes a request.

Note that in the preceeding diagrams, C's address space will actually contain C's stack and the I/O area may be on the stack or on the heap. Similarly, A's service capability is probably on A's stack.

The service call algorithm is given below. In this algorithm, 'current process' is an indirect pointer to a psv. It is added to or subtracted from to move it through the execute stack.

```
Request (level_diff, service_no)
  current_process := current_process + level_difference;
                      (pop execute stack)
  switch contexts from user to server stack
  current_process^.execute_link := NIL;
  use user.top_of_stack to index server's service list
  pop user stack
  execute server code for parameter construction
                      (constructs addresses of actual parameters on stack)
  For each service parameter (copy actual parameters)
    copy parameter on user stack to parameter^ on server stack
    pop user stack
  For each service parameter (copy revocation pointers)
    push parameter on server stack onto user stack
  ...call server procedure...
```

5.3 Execute Implementation.

The code that implements the Execute sequence 'undoes' the service request, then leaves the processor executing the user's code. This requires that execute links be used to reconstruct the execute stack, and that the

service procedure's actual parameters be revoked:

```

Execute (psv_cap)
  current_process^.execute_link := psv_cap;
  while psv_cap^.execute_link <> NIL do (rebuild execute stack)
    push psv_cap on execute stack
    psv_cap := psv_cap^.execute_link;
  push psv_cap onto execute stack;      (this process gets the processor)
  switch contexts from server to user stack
  for each service parameter
    parameter^ := NIL;                  (revoke parameter capabilities)
    pop user stack;
  ...continue executing user code...

```

6 Conclusion.

The Execute statement is a high level means of controlling instantiation and management of processes whose number and requirements are unknown at compile time. Together with capabilities, the Execute statement extends the benefits of high level languages and strong typing across operating systems and user programs, thus simplifying proving and verification. This is done efficiently and without additional hardware. The Execute statement explicitly specifies the interface between conceptual levels of abstraction. Static language constraints provide inter-level protection.

In this paper, the operating system structures supported by the Execute statement have been discussed. Language constructs and examples of their use have been shown, and implementation algorithms have been described. In [McKendry, 81], a more detailed description of the statement is to be found. Future research includes the issues of service parameter type import and export, the question of loader-correctness, implementation of system services such as interrupts and time allocation, and multiprocessor implementations of the Execute statement.

A pilot implementation of capabilities and the Execute statement based on Path Pascal [Campbell & Kolstad, 80] is currently under construction for the Prime 650.

7 Acknowledgement.

This work was undertaken as part of M. S. McKendry's Ph.D. Thesis research.

8 References.

- [Brinch Hansen, 75] Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Trans. on SE., Vol. SE-1, pp. 199-207, June, 1975.
- [Campbell & Kolstad, 80] Campbell, R. H. and R. B. Kolstad, "An Overview of Path Pascal's Design," Sigplan Notices, Vol. 15, No. 9, pp. 13-14, September, 1980.
- [Dijkstra, 68] Dijkstra, E. W., "The Structure of the THE Multiprogramming System," CACM, Vol. 11, No. 5, pp. 341-346, May, 1968.
- [Horning & Randell, 73] Horning, J. J. and B. Randell, "Process Structuring," Computing Surveys, Vol. 5, No. 1, pp.5-30, March, 1973.
- [Jensen & Wirth, 74] Jensen, K., and N. Wirth, Pascal User Manual and Report, Springer-Verlag, 1974.
- [Liskov, 72] Liskov, B. H., "The Design of the Venus Operating System," CACM, Vol. 15, No. 3, pp. 144-149, March, 1972.
- [McKendry & Campbell, 80] McKendry, M. S., and R. H. Campbell, "Capabilities for High Level Languages," University of Illinois, Department of Computer Science, TR UIUCDCS-R-80-1039, December, 1980.
- [McKendry, 81] McKendry, M. S., "Mechanisms for High Level Language Operating Systems," Ph.D. Thesis, University of Illinois at Urbana-Champaign. In preparation.
- [Morris, 73] Morris, J. H., "Protection in Programming Languages," CACM, Vol. 16, No. 1, pp. 15-21, January, 1973.
- [Neumann, et al., 77] Neumann, P. G., et al., "A Provably Secure Operating System: the System, its Applications, and Proofs," Stanford Research Institute Project 4332, Final Report, 1977.
- [Wirth, 77] Wirth, N., "Modula, A Language for Modular Multiprogramming," Software Practise & Experience, Vol. 7, pp. 3-35, January, 1977.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-80-1044	2.	3. Recipient's Accession No.
4. Title and Subtitle THE EXECUTE STATEMENT: DESIGN, EXAMPLES, AND IMPLEMENTATION ALGORITHMS		5. Report Date December 1980	
		6.	
7. Author(s) Martin S. McKendry and Roy H. Campbell		8. Performing Organization Rept. No. R-80-1044	
9. Performing Organization Name and Address Department of Computer Science University of Illinois Urbana, IL 61801		10. Project/Task/Work Unit No.	
		11. Contract/Grant No. NASA NSG 1471	
12. Sponsoring Organization Name and Address NASA Langley Research Center Hampton, VA 23665		13. Type of Report & Period Covered technical	
		14.	
15. Supplementary Notes			
16. Abstracts The Execute statement is a language mechanism that manages the interface between levels of processes in level structured operating systems. It is a high level means of controlling instantiation and management of processes whose number and requirements are unknown when the operating system is compiled. The statement facilitates implementation of services such as memory management, interrupt management, and synchronization. Together with language capabilities, the Execute statement extends the benefits of high level languages to entire operating systems. This is done efficiently, and without additional hardware. The Execute statement makes explicit the interfaces between conceptual levels of abstraction, with inter-level protection enforced by static language constraints.			
17. Key Words and Document Analysis. 17a. Descriptors operating system language process management protection language capabilities			
17b. Identifiers/Open-Ended Terms 			
17c. COSATI Field/Group 			
18. Availability Statement unlimited		19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 26
		20. Security Class (This Page) UNCLASSIFIED	22. Price

